Assignment -2 Report submitted for
Artificial Intelligence (UNC504) by

Name of student: Prabhnoor Singh

Roll number: 102115059

Group: 3NC3

Submitted to

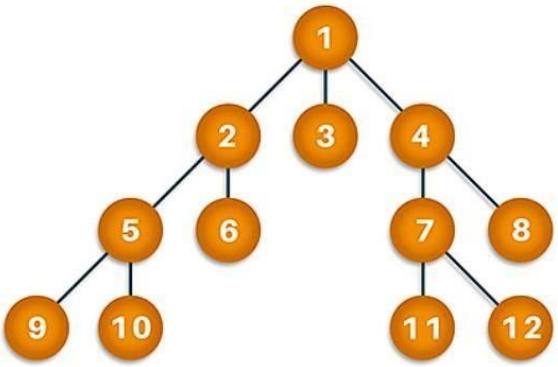
Dr. Tanvi Dovedi

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY, (A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB
INDIA

July-Dec 2023

Assignment 2:

Ques1: For the given tree below, write Python code to find the tree traverse and path from start node (1) to goal node (8) using breadth first search (BFS) and depth first search (DFS).



Algo:

(a) Breadth First Search:

1. Create an empty set visited to keep track of visited nodes.
2. Create a queue queue and enqueue the start node.
3. While the queue is not empty:
   a. Dequeue the front node from the queue and mark it as visited.
   b. If the dequeued node is equal to the end node, print it and return.
   c. Print the dequeued node.
   d. For each neighbor of the dequeued node in the graph:
      i. If the neighbor has not been visited, enqueue it.
4. If the loop completes without finding the end node, it means there is no path from start to end.

(b) Depth First Search:

1. Mark the start node as visited by adding it to the visited set.
2. Print the start node.
3. If the start node is equal to the end node, return.
4. For each neighbor of the start node in the graph:
   a. If the neighbor has not been visited:
      a. Recursively call the DFS function with the neighbor as the new start node.
5. If the loop completes without finding the end node, it means there is no path from start to end.

Code:

(a) Breadth First Search:

```
def bfs(graph, start, end):
  visited = set() queue
  = [start]

  while queue:
    node = queue.pop(0)
    visited.add(node)
    if(node == end):
        print(node, end=" ")
        return
 print(node, end=" ")

    for neighbor in graph[node]:
      if neighbor not in visited:
        queue.append(neighbor)

tree = {
  1: [2, 3, 4],
  2: [5, 6],
  3:[],
  4: [7, 8],
  5: [9, 10],
  6:[],
  7: [11, 12],
  8: [],
  9: [],
  10: [],
  11: [],
  12: [] }
bfs(tree,1,8)
```

(b) Depth First Search:

```
def dfs(graph, start, end, visited=set()):
  visited.add(start)
  print(start, end=" ")

  if start == end:
    return
 for neighbor in graph[start]:
    if neighbor not in visited:
      dfs(graph, neighbor, end, visited)

tree = {
  1: [2, 3, 4],
  2: [5, 6],
  3:[],
  4: [7, 8],
  5: [9, 10],
  6:[],
  7: [11, 12],
  8: [],
  9: [],
  10: [],
  11: [],
  12: [] }
dfs(tree,1,8)
```

Output:

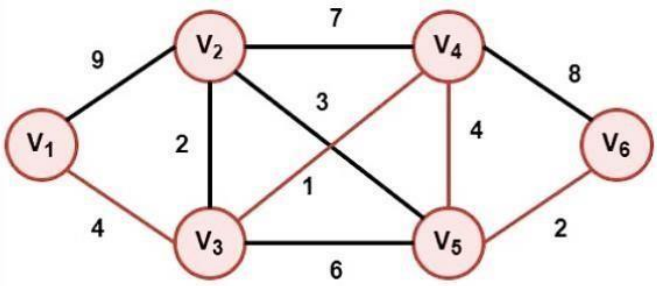
(a) Breadth First Search:

```
1 2 3 4 5 6 7 8
```

(b) Depth First Search:

```
1 2 5 9 10 6 3 4 7 11 12 8
```

Ques2: For the given graph below, write the Python code to find the lowest path from start node (V1) to goal node (V6) using uniform cost search.



Algo:

1. Initialize an empty priority queue (min-heap) called queue. Each item in the queue is a tuple (cost, node, path) representing the cost of the path to reach node and the path itself. Initially, add (0, start, [start]) to the queue, indicating that the cost to reach the start node is 0, and the path contains only the start node.

2. While the queue is not empty:
   a. Pop the node with the lowest cost from the queue. Let's call this node node, and its path path.
   b. If node is equal to goal, you have found a path: Return the cost as the cost of the path.
      Return path as the sequence of nodes in the path.
   c. If node is not equal to goal, continue with the search.
   d. For each neighbor of node (neighbors are obtained from adjacency_list.get(node, [])), do the following:
      i. Calculate the new_cost by adding the cost of the edge from node to the neighbor to the current cost.
      ii. Create a new_path by appending the neighbor to the current path.
      iii. Push (new_cost, neighbor, new_path) into the queue. This represents exploring the neighbor with the updated cost and path.

3. If the queue becomes empty and no path to goal is found, return "No path found."

Code:

```
import heapq

def uniform_cost_search(adjacency_list, weights, start, goal):
    queue = [(0, start, [start])]
 while queue:
        cost, node, path = heapq.heappop(queue)

        if node == goal: return
            cost, path
```

```
        for neighbor in adjacency_list.get(node, []):
            new_cost = cost + weights[(node, neighbor)] new_path
            = path + [neighbor]
            heapq.heappush(queue, (new_cost, neighbor, new_path))
 return "No path found"
 adjacency_list = {
     1: [2, 3],
     2: [3, 4, 5],
     3: [2, 5, 4],
     4: [2, 3, 5, 6],
     5: [2,3,4,6],
     6: [4, 5]
 }
 weights = {
     (1, 2): 9,
     (2, 1): 9,
     (1, 3): 4,
     (2, 3): 2,
     (3, 2): 2,
     (2, 4): 7,
     (4, 2): 7,
     (2, 5): 3,
     (5, 2): 3,
     (3, 5): 6,
     (5, 3): 6,
     (3, 4): 1,
     (4, 3): 1,
     (4, 5): 4,
     (5, 4): 4,
     (5, 6): 2,
     (6, 5): 2,
     (4, 6): 8,
     (6, 4): 8
 }
 start = 1 goal = 6 print(uniform_cost_search(adjacency_list,
weights, start, goal))
```

Output:

```
(11, [1, 3, 2, 5, 6])
```